

Title of Invention:

Implied Instruction Set Computing (IISC) / Dual Instruction Set Computing (DISC) / Single Instruction Set Computing (SISC) / Recurring Multiple Instruction Set Computing (RMISC) Based Computing Machine / Apparatus / Processor

1 Technical Field

[1] This invention related to the technical fields of Computing Machinery, as well as Electrical and Electronic Engineering; and more particularly to the fields of Computer Organisation and Design, Computer Architecture and Compilers and related Technologies.

2 Current State Of The Art

2.1 Background Art

[2] Currently, there are numerous designs, organization and architectures for processors. Out of these, there are two predominant processors architectures and processor organization design methodologies. These two architectures are namely: Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). These architectures are now being superseded by newer architectures (like Explicitly Parallel Instruction Computing - EPIC) due to their limitations. Also in recent times there has been interest in dynamic architectures where the parts of the processor are loaded as needed. Architectures which use Very Long Instruction Words (VLIW), Tagged VLIW, Explicitly Parallel Instruction Computing, etc. which use different classes of instructions are packaged together have sprung up, i.e., each class of instruction is sent to different units of the processor or FU for parallel execution.

[3] The limitation of the existing architectures will be addressed by this invention. The section that follows will discuss in detail these technical problems which have been completely or partially overcome by this invention.

Technical Problem:

2.2 Disadvantages, Short Comings and Drawbacks in the Prior Arts

[4] In the present state of art with regard to computing is such that a processor in a computing machine/apparatus fetchers an instruction decodes it and then executes it until such time that the processor is instructed to halt. Despite research and development into processor improvement, the decode and execute phase is complex and consumes time and power. Simplification of these phases is in deed welcome. Moreover, there are possible improvements in the “instruction fetch” phase, which is further elaborated below.

[5] Many of the existing electronic computing machinery suffers from the limitations of the Von Neumann Architecture to varying degree, despite recent effort to overcome these limitations; i.e., memory I/O bandwidth has become a limiting factor in the processor. A reduction in the overall executable code size will help in overcoming memory Input / Output (I/O) bandwidth problem.

[6] Generally, a processor (herewith referred as a Processing Entity - PE) can execute only a single instruction at a time. There has been some effort to provide mechanisms to execute more than one instruction (belonging to different instruction class) at a time. These schemes have limitation in the number of possible parallel instructions. In addition, there are work-a-rounds like pipe lining where different parts of an instruction are executed in parallel. These mechanisms are very complex in terms of hardware, which leads to power dissipation, and result in some additional overhead during processing as well. This processing overhead is a waste of computational power in the processor and does not add value to the computational task executed. Making instruction parallel as possible (with minimal restrictions) will increase performance. Moreover, elimination of many complex parallelising schemes will reduce the component (transistors, etc.) density and heat dissipation. On the other hand, the saved components can be used to enhance the processor capabilities.

[7] Currently, the processor designs are not in terms of loosely coupled modular units, which facilitated extensibility of processor functionality. Due to this, extensions to the existing processors to provide additional functionality when newer versions are produced will need substantial reworking in terms of the design as well as testing. These issues can be partially overcome by using VLIW techniques, within the premise of the prior arts; but still there is room for improvement. There is substantial research in dynamic processing architectures though such processors are not commercially wide spread.

[8] In VLIW and related architectures (like EPIC), the instructions are in “packets”, which allows room for a shorted (more optimal) instruction size than RISC and CISC based architectures where the instructions are generally at least as long as the word size of the processor. Though VLIW architectures have certain advantages and are gaining popularity, they architectures suffer from code bloat, which results from the use of NOOP (no operator) instructions when there is no instruction to be sent to a functional unit.

[9] When considering the instruction and processor architectures in the prior arts, the main emphasis is on the operations the PE is to perform. In reality, a programme is data undergoing transformations by the operations performed on it.

[10] Threading can be used to programme section of parallelism in algorithms but they do not achieve instruction level parallelism, i.e., there still could be a number of operations in the threads which can be executed in parallel.

[11] To summarise the implications of currently existing design is as follows:

- Decoding phase of instructions are complex, time consuming and leads to more power dissipation (in pipe lining dependencies must be identified before execution and in trying to execute instruction in more than one class a number of instructions should be examined before the execution.)
- Execution follows decoding of an instruction (which consumes more time as a whole.)
- Fetching instructions is less efficient since the cache might not be directly manipulatable.
- Processors are not designed to gain full advantage from instruction level parallelism.
- Memory I/O Bandwidth is effectively wasted in:
 - Transferring NOOP instructions.
 - Use of instructions size which is larger than needed (instructions are generally word aliened.)

3 Theoretical Foundation

[12] Currently a computational task is generally specified using algorithms. Algorithms are limited in the ability to show operation and dependencies between them. (They generally show the operation as a series of steps, conditional branching, and repetition.) But a computational can be alternatively represented using Graphs as discussed below.

[13] This invention used the concept of Dependency Graphs. A Dependency Graph is a digraph (like network diagram) where vertices represent operations and the edges represent the Data Dependency between operations. The arrows of the diagram appear from left to right as in a network diagram. Time flows in the diagram is from left to right.

[14] In the case hypothetical the above described configuration is ideal, but in the case of a real world implementation. The above graph needs modification. Slots will be introduced representing the maximum concurrent operation that can be executed.

[15] A slot will at least represent an operation. A block instances will represent the blocks across time. Even in such a case the dependencies are modelled as edges as before.

[16] Slots can be partitioned or blocked. The blocks can be a Write Block or a Read Block or both. A Write Block can only hold an operation which is a Write Operation and a Read Block can only hold a Read Operation. A Write Operation sets the input of an operation. If the Write Block is used then at least one Write Operation should precede an operation. If a Read Block is used at least one Read Operation should follow an operation. These special operations can be viewed as moving information needed for operations.

[17] In case a Read Block is used the Read Operation should reside in a previous Block Instance and if a Write Block is used the Write Operation should be used in a Block Instance after the Block where the Operation resides.

[18] The crux or essence is that there is a network of operations where data flows through operations, which transform it. This theory / computation model is inspired the invention described below. Using this scheme more instruction level parallelism can be gained.

Disclosure of Invention:

Technical Solution:

4 Description of Invention:

[19] The mode of practicing the art of the invention and its variation as contemplated by the inventor would be presented herewith. The advantageous effects of practicing the art of the invention as described thereafter, the embodiments contemplated in practicing the art of the invention, the best mode contemplated by the inventor at the time of writing in the shoes of a person with average skills in the art, the manner in which the art of the invention should be practiced as appearing to the inventor at the time of writing, the applicability of practicing the art of the invention as contemplated, would follow.

4.1 Overview: a summary of the invention

[20] The computing apparatus/computing machine/processor described in this invention will be such that, it will be able to achieve a computational task by moving data to and from units which carry out computational tasks. At the time of writing, considering the current state of the art, for illustration purposes only, it is contemplated that the input can be considered to be in the form of registers, which are wired to FU that carry out a computational operation and produce various outputs which are also registers; but this scheme may be varied to suite new developments and varying requirements by a person skilled in the art, in practicing the invention. Registers are embodiments, which hold in memory a piece of data. Any embodiment which serves the above said purpose would be here with referred as a register.

[21] In the current invention, the processor is designed to ensure that the architectural emphasis is not merely on operations performed by a PE. The emphasis is on the flow of data through various operations carried out by the PE.

[22] The invention will employ embodiments called Functional Units (FUs) which will perform a computational operation. The operands of the operation performed by the FUs will reside in input registers. The output will be placed in Output Registers.

[23] The purpose of 'instruction' is to route values residing in registers as input to the FUs. These instructions are implied (i.e., they have a fixed number and they appear in pre-determined order) thus they can omit any operands, which instruct the Register Data Bus Controller (also called the Data Bus for short in the description) to move the registers.

[24] Many of the registers will be wired to FUs. When written to, the FUs will produce an output, which is placed in Output Registers that are read only.

[25] Some of the connection between the registers and FUs may be pre-established, some would be established dynamically during boot time or perhaps later on, but this operations are not frequent. Some FUs will be pre-fabricated (flashed on to the chip) whereas some would be dynamically loaded. FU loading operations are also not too frequent.

[26] FUs are embodiment that can be practiced with the invention. These embodiments many not be limited to the once presented here. A person skilled in the art will recognise what additional embodiment to practice, what embodiments described here needs customising and what embodiments to be left out, in practicing the art of the invention.

[27] A more detail discussion of the architecture in the novel apparatus will follow:

4.2 Design Objectives at a Glance

[28] An overview of the design objectives are as follows:

4.2.1 Issues Addressed By the Solution:

[29] This proposed solution will try to address the following areas:

- Reduction in complexity of decoding an instruction.
 - Reduction of electronic devices (e.g. logic gates, transistors, etc.) used for decoding.
 - Reduction of power dissipated.
 - Reduce total time spent in the decode plus the execute phase in a PE.
- Much simpler decode phase or elimination of the phase, i.e., removing the necessarily of an explicit decode.
- Instruction level parallelism is taken advantage to the maximum possible extent.
- More control of the internal working of the processor to facilitate optimization by a compiler.
- Enabling easier extensions to existing design with minimal reworking and testing.
- Smaller instruction size and mineralising the need of certain instruction (e.g. NOOP) to reduce bandwidth of data transfer.
- More control to the compiler in order to facilitate better and more optimal code generation.

[30] In order to achieve the above the processor organization and architecture the follows, is proposed. Initially and over view is given followed by a further description.

4.2.2 Brief Descriptive Overview of the Processor Organization in Pursuant of the Design Objectives:

[31] In order to achieve the above the processor would be organized as in the following brief overview:

- It would have special purpose registers.
- These registers would be hard-wired or pre-connected to Functional Unit (FU).
- The FUs might not have internal/hidden/shadow registers.
- The execution would be implicit.
- The processor can only execute only one, two instructions or more instructions where one instruction will always be followed by the other, forming a recurring patter of instructions.
- The input of these FUs can be classified, for convenience, as Input Registers and the output of the FUs as Output Registers. The Input Registers will form

the basis of input for a FU and the Output Registers will be where the information, after an operation, is written.

- Since an instruction will always be always ordered as a pair of the two executable instructions, the instruction can be made implied, i.e., the processor does not need any op code to identify the instruction.
- The compiler will be to identify existing information available (probably residing in Output Registers) and then would target that information to other Input Registers, i.e., registers will be selected and the content is targeted to Input Registers.
- In the normal cause of operation, the following will take place in the PE:
 - The registers are selected as mentioned above.
 - They are targeted to Input Registers.
 - The Input Registers are hard wired (or pre connected) to a unit, which performs a logical operation on the input data.
 - The output of the logical operations are hard-wired (or pre connected) to Output Registers. Therefore, they will contain the results of the logical operations preformed by the unit.
 - Parts of the processor which are used will be known advance so these parts can be activated and deactivated as needed to save power. If the processor is implemented in a technology where the logic can be dynamically changed, it is possible load an unload functionality as and when needed.

4.2.3 The Instructions used for the Processor:

[32] Instruction Set Architecture (ISA) emphasizes data and its dataflow through a series of operation which transform the data rather than an emphasis on the operators and the data as operands as in the current context. The purpose of the instructions is to identify registers which needs to be moved and to where to move them. Since the identification is always followed by the movement of those values into the target registers, these instructions can be implied, i.e., the processor will not need any instructions in its instruction set.

4.2.4 The Layout of Functional Units or Operation Mills:

[33] The Layout of operation mills (similar to FU in VLIW but can only carry out a few simple operation without in order to avoid the need of decoding) would be as a series of independent units which receive input from a sub set of the machines registers and the output is hard-wired (or pre connected) to another subset of registers. i.e., maps values in some registers to another subset. Neither the registers involved nor the instruction needs to be decoded for this to happen.

4.3 Main Topological and Architectural Characterises

[34] In this architecture, there are a series or an array of FU and a larger number of registers. A FU has certain number of connectors where input and output connection between the FU and registers can be pre-established. Connections are made between the registers and FU during the processor initialization, for "special purpose" processors these connections can be pre-fabricated. In any case some of the connection should be pre-fabricated, especially the ones used during the processor initialization.

[35] As depicted in the drawings ‘Main Architecture’ and ‘Alternate Architecture’, two topologies are possible. In one topology, a FU has many inputs, as well as, outputs. The other topology is that there is only one output for a FU. In the latter topology, the number of output connectors can be restricted to one. In the first topology there is a possibility to group the outputs of the FU each other are easily deducible when computing the other. E.g. when computing $A - B$ or $B - A$ where A and B are input registers the Output Registers may be filled with $A - B$, $B - A$, $-A$, $-B$, $\sim A$, $\sim B$ where \sim is negation of the bits in the register.

4.3.1 Functional Unit Arrangement and Physical Hardware Topology

4.3.1.1 Purpose and Layout of FU

[36] The FUs are organized in an array. The data exchanged between the registers are routed using the Data Bus. A FU has input from Input Registers. The output is then saved in the Output Registers. The values in these registers are moved around using the Data Bus.

4.3.1.2 FU Initialization: loading and establishing connections

[37] During the processor unitizations the FUs will need to be loaded and any input and Output Registers will be bound. Some of the FU will be prefabricated and the connections between these FUs and the Input Registers will be pre determined. This will be necessitated since there should be basic FU like the Data Bus, Memory Management, Catch Management, and FU loading FU to start up and initialize the processor and load the optional FU. There also could be arrangements where FU may be prefabricated but not the connections. It is possible that some of the FU will have some registers wired which cannot be changed. This arrangement will have a saving on the components, which would otherwise be needed in dynamically binding registers.

4.3.1.3 Activation of a FU

[38] A FU will carry out the computational task in synch with the PE's clock. There are two possible methods contemplated to activate the computational task performed by the PE:

- Compute only the output from the Input Register regardless of any input given,
- Compute it only when the Input Registers are updated.

The later is more power efficient though some additional logical components and processing overhead are necessitated.

4.3.2 Boot-up Process: setting the connections

[39] In the boot-up process instructions from the ROM is sent to the processor's (here by referred as Processing Entity - PE) cache. The execution within the PE starts at the first instruction of the cache after it is loaded.

4.3.2.1 Role of Prefabricated Pre-Connected / Hard-Wired FU

[40] The prefabricated and pre-connected FU would load subsequent optional FUs into the processor. In this process, the connections of the FUs are also established. A more detail discussion can be found above.

4.3.3 Instruction Alignment within a Word

[41] Data parameters will occupy a word or a part of it, or more. Instructions would be moved into the PE for execution as packets.

4.3.4 Register Layout

[42] In the proposed architecture, there will be a large quantity of registers. These registers will be connected to the Data Bus using connectors. As previously mentioned some of the connectors will be prefabricated. The connectors are of two types:

- Input connectors, and
- Output connectors.

4.3.5 Multi PE Architecture

[43] A system may have multiple PE of which the Memory Controller is a dedicated PE, which transfer memory objects between memory, and PE. The PE in the system interacts with the memory controller in a homogeneous manner, but the PE themselves may not be heterogeneous.

4.3.6 Alternate Architecture

4.3.6.1 Limiting the Output registers to One

[44] In the alternate architecture as depicted in figure labelled 'Alternate Architecture', a FU has only one output. This would result in more compact FUs and the connections would also be simplified in terms of hardware and the components.

4.3.6.2 Multiplicity of Instructions

[45] This invention involves using a single instruction, two instructions or more instruction, which appear in recurring patterns. Since the instructions are in pre-defined patterns, the instructions can be deduced meaning that they are implied.

[46] In the case that there is only a single instruction, it would be a single move. The move instruction is effectively choosing a register and setting the target register with its value, which can be viewed as two instructions.

[47] In this invention, a recurring block of instructions can be viewed as a single instruction or as smaller set of instructions. E.g. if the operations are schedulable viewed as:

- The register is selected for moving,
- The targeted register is selected for the data to be placed, and
- Time in which to activate the FU.

The combination of the last two can be taken as a single instruction or the whole as a single 'move and set time' instruction.

[48] The draw back with the above scheme is that the FUs schedule gets updated with every move. In the case that a FU accepts more than one input, the schedule is unnecessarily updated more than once. Since a FU generally will have two inputs (operands) then the instruction can be two 'move' ('select register', 'write register') followed by a set time instruction. The set time may be associated with the last or first FU involved – the two moves may involve different FUs. In the case, were a FU takes

only a single input, a dummy ‘move’ or perhaps a ‘move’ to partially set the inputs of another FU will need to be inserted. This is very complex juggling for the compiler. Also if the set timer instructing may not be needed that often.

4.3.6.3 Alternate Modes of Practicing the Art of the Invention

[49] E.g. if we consider that the invention is practiced in a machine with limited memory and resources (like an embedded environment but this mode of practice is not restricted to such environment and a person skilled in the art will recognise where it can be practiced.)

[50] In such a environment the instructions can be arranged as recurring multiple of the follows (for illustration purposes only):

- Load item from memory to Input Register 1,
- Load item from memory to Input Register 2,
- Store item from Output Register to memory.

[51] In order to practice this alternate form a FU should have a known topology. E.g. one or two inputs with an output. (In case some FUs take only one input then a dummy load may be occasionally needed.)

4.4 Instruction Layout and Assembly Language

[52] The instructions are organized as pair of selecting a register and targeting the content of the selected register to another destination register. The instructions are taken into the processor in chunks or packets. These instruction packets are executed together, i.e., they are executed in parallel. The instructions packed together should be such that Input Registers used will not get overwritten, otherwise parallel execution may not be possible.

[53] The operations can be timbale. In a case the instruction would either with a timing operand would specify the timing/schedule of a FU or the FUs will have a timer Input Register while the instruction does not contain a timing operand. In the latter case, these timers will need setting up by moving values in to timer registers using other operations/instructions. The previous case involved:

- The register selected for moving,
- The targeted register for the moving, and
- Time in which to do the move.

This can be considered as one instruction:

- Move and set time.

This can be further decomposed as:

- Move
- Set time

or

- Select
- Target and set time.

These instructions can be further decomposed into:

- Select
- Target
- Set time

which would all ways appear in this order. Since there are no op code involved and all the operands that are sent to the Data Bus Controller are control parameters in can be considered that this approach does not use any instructions at all. The relative merits of the two approaches will appear in the Best Mode Disclosure.

4.4.1 Handling of Dependencies in Instructions

[54] Instructions have dependencies between the registers. A set of instructions, which can be executed independently in a given time (clock cycle), is considered in the same level. Instructions are packed together as chunks, which are transferred to the processor at one.

[55] There would be cases where the instruction in a given packed may not be in the same level (needs to be executed in different clock cycles). This case can be handled in many ways:

- Iteratively execute the packet equal to the number of Level Breaks in a Packet. In the initial execution some of the Input Registers will receive garbage the values which are computed within the packet. This is remedied by repeatedly executing the instructions where the results of the previous phase become available. Repeated execution can be implemented as follows:
 - The FU handling execution will repeatedly execute the Instruction Packet, or
 - The Instructions Packets will be duplicated in the Instruction Pipe.
- A second option is to stall the processor then there is a dependency level break. This would be complex in terms of hardware.
- The other is to pad the packet with NOOPs. This can be done in two ways:
 - At the compiler stage,
 - The packet is padded, in hardware, in the fetching process opposed to padding by the compiler at compile time. This option will result in lower code size and would save memory Input / Output (I/O) bandwidth. This is dealt in the Code Level Management FU.

[56] A special FU will be available to specify exactly where in the code the Dependency Level breaks occur. These will be stacked and later used to make adjustments in the instruction pipeline as mentioned below.

[57] Alternatively, if all instructions were schedulable then the ability to specify and handle Dependency Levels is not needed. The instruction will be just scheduled to be executed after a certain laps of time – ideally measured using clock cycles. If instructions in an Instruction Packet are in different levels then those instructions belonging to later Dependency Levels can be scheduled to be executed later as deemed appropriated by the compiler. (Effectively the above schemes of handling dependencies – in the case where not all the instruction are schedulable – do a rescheduling of the instructions.)

4.4.2 Instruction Pipelining

[58] The instruction are executed are queued up in a pipeline. In queuing process, the instruction packets may be adjusted or other special transformations are carried out. The adjustment takes place if there is a Dependency Level Breaks. E.g. the adjustments would include: duplicating Instruction Packets, break Instruction Packets and insert NOOPs, etc.

4.4.3 Array Registers / Bit String Registers / Byte Buffer Registers / Very Long Registers

[59] In practicing the art of the invention, FU will have Bit Buffered/Bit String/Very Long Bit registers. They would occupy number of words. Using these registers and FU to which they are connected to, large calculations like vector multiplication, summing a large list of values, changing case of a string, etc. can be carried out in a single step.

[60] When accessing and addressing these special registers two methods can be employed:

- The special register and the offset of words in the long registers can be sent a special FU to extract the value.
- For the purpose of addressing a long register will appear as many shorter registers.

4.5 Memory Management

[61] The memory management in this architecture is done in two phases. First, the internal Cache Management Unit is accessed. This unit further delegates this to the external Memory Controller. The external memory controller is a separate dedicated PE, which focuses in moving chunks of memory to the cache of the PE.

4.5.1 Explicit Cache Management

[62] One of the innovative features of this is Explicit Cache Management is that the compiler can control the cache. The compiler is most close to the code. It is in a position to optimize the use of cache since the compiler knows which of the variables are used and for which length they are kept in memory.

[63] The cache is partitioned into two parts:

- One part is for instructions, and
- The other is for data.

[64] The advantage of having two caches for data and instructions will be felt in the design of the call and stack management functionality. The data partition of the cache is addressable through a moving window depending on the context.

[65] The Cache Management unit delegated the responsibility to the Memory Controller to interact with memory.

[66] The Memory Management FU moves cached data to the registers and register data to the cache.

4.5.2 Instruction Fetching and Execution

[67] The Memory Controller moves the data from the memory to the cache. The Memory Management FU moves the needed instructions to the Data Bus Controller, which execute the instructions. The Cache Management FU liaisons with the Memory Controller, which manages the memory of the whole system, to move data to and from the cache. The instructions in the cache are executed in order unless the looping and branching FUs do not intervene. The cache is implemented in a circular manner. After the boundary of the cache is reached the start point of execution is set to the

beginning, i.e., the Instruction Pointer is reset to zero. It is the responsibility of the compiler to load instructions to the cache so that the code executed is meaningful.

[68] Before execution the instructions are queued in a pipe-line. In the queuing process adjustments are made if the architecture requires that code is adjusted by inserting NOOP instructions.

4.6 Booting

[69] In this invention, only instructions in the cache are executable. Therefore, at the time of booting the instruction that need execution should be transferred to the cache of each PE.

[70] The required data should also be at a well know location to which the processor would be configured to look at by default or by the intervention of an external controller.

4.7 Stack and Call Management

[71] The stack of a PE is implemented internally. On calling the processor switches the data cache and register context, i.e., the window on the addressable part of the cache, and the registers will change. In this process parts of the cache which was addressable will become un-addressable. In order to transfer data between functions it would be ideal that a part of the cache overlaps in this process.

4.8 Processor Context Switching

[72] The hardware support for process context switching is carried out in the process management FU. The process context switching can be carried out using two approaches:

- In the first approach when the process context is switched the current context the stacked. The shared part of the context is passed is stacked in a separated stack which has information about the process context is saved. The process context can switch to a newer process or can switch back to the previously executed process. When the stacks are filled they are spilled into memory. On switching back the stacked memory context should be spilled into memory.
- In the second approach a process table is maintained. A process context switch is carried out by coping the process context into this process context table. The reverse is done to load it. This approach requires a little bit more memory than the above since the stack is replicated between the contexts switched but will be faster than the above since the memory is not too often accessed.

4.9 Interrupts

[73] Interrupts are of two types:

- Software interrupts, and
- Hardware interrupts.

[74] Software interrupts are raised by loading the interrupt number along with the parameters into the Input Registers of the FU.

[75] Alternatively, this architecture can be implemented without interrupts. There could be FU, which perform the interrupt request. In parallel to the normal operations

of the processor. Software interrupts can be accommodated FU slots with reprogrammable logic.

4.10 Branching and Looping

[76] Like may PE this also provide the functionality of looping and branching but in a novel manner. Looping and branching instructions can be scheduled, i.e., the number of clock cycles before an instruction is executed can be specified when the instruction is processed.

[77] Branching instruction can be of types:

- Repetition for a fixed number of executions,
- Repetition while a condition is true.

[78] In the case where fixed number of executions are possible, the instruction is scheduled with the number of repeated. After the number of cycles the block of code is repeatedly executed. The count is updated for each execution. This instruction is made schedulable since the repetition count may need computing and may take time before it is known. This delay can optimally used to execute initialization instructions. The clock count can only be updated once until all repetition is executed. The purpose of doing this is to ensure that the instruction fetch unit can fill the instruction pipe line with the appropriate instructions.

[79] The purpose of the conditional looping FU is to repeat the number of instructions while a condition is true. The instruction is schedulable since the condition may take time to compute. When computer this Input Register is moved to the condition. But the clock cycle counts start when it is first accessed. The condition is write once per execution. This is to ensure that the instruction flow can be predicted in advance to fill the pre-fetched instruction pipe-line.

[80] The branching uses a similar schedulable approach. The delay for when the condition is true and false can be different. When the counter of clock cycles runs down to 0 in one of the counters the condition is checked and the branch is executed if appropriate. Otherwise the instruction execution is delayed until the other counter runs down to 0 and the appropriate branch is taken. The condition is only write once for a given execution.

[81] In a case where an instruction is write once. The specific implementation can decide what to do when the write once register is written multiple times. Following are some of the possibilities:

- The write can be ignored,
- An interrupt can be raised.

[82] Once the condition is known the pre-fetched instruction pipe-line is filled when the appropriate instructions from location where control is transferred to. If the clock cycles count before the jump is long enough the pipeline is filled smoothly, i.e., the processor will not need to stall. In this arrangement the instruction pipeline does not need to be emptied due to jumps. When there are no pre-fetched instructions the processor stalls until the pre-fetched pipeline is filled.

4.11 Time Slicing

[83] In order to run multiple processors in parallel time slicing is needed. It is proposed that this is implemented in hardware. The time slicing FU deals with the process context switching.

[84] The Operating System (OS) may only access the process context switching instruction.

[85] In the case where the stack is used for process context another FU will be used to switch back.

4.12 Thread Scheduling

[86] The executing threads will be executed using Thread Scheduling FU. A table of threads will be maintained in the part of the cache. Parts of this cache may be switched in and from the memory. The active thread is marked in the table. This pointer is moved in circular manner.

4.13 Compiler

[87] For the purpose of this implementation, the compiler will need to model the operation in terms of a Dependency Graph, where the operations are modelled as vertices and the dependencies are modelled as edges. From left to right the graph will denote the time lapsed or operations performed. Operations that can be carried out at a given time are considered to be at the same Dependency Level.

[88] The operations are matched with the FUs which are capable of executing the operations. Depending on the availability of FU and dependencies the instructions are packed into packets; these packets are the executables. The code is arranged to optimally use the FU and to maximize parallelism among instructions.

[89] A Dependency Graph adapted to suit the availability of FUs can be called a Slotted Dependency Graph. This graph will have an additional dimension of slots which represent the available FUs at a given time slot. If the slots of a particular type are less than the number of possible operations at a given Dependency Level, then some of the operations will need to be moved to higher levels while considering the overall efficiency of the programme. The one getting promoted could be the operation on promoting will not have a domino effect of promoting of other operations if such exist or an operation which will lead to lesser subsequent promotions. A person significantly skilled in the art of Computer Programming will be able to recognise the best approach on how to practice this, whilst maintaining the essence presented here.

4.14 Compiling High Level Language (HLL)

[90] Many of the HLLs and algorithmic constructs can be reduced to logical constructs. The FUs are implemented using logic gates and should be optimal to minimize execution time. The logic of the FUs would be specified using a special purpose HLL.

5 Advantageous Effects

[91] This invention would make significant progress over the prior arts due to its numerous advantageous effects. The advantages can be classified under the following:

- Advantages in the instruction fetch over current processor/computing machine designs.
- Advantages in the decode phase over existing processor/computing machine designs.
- Advantages in execution over other existing state of technology.
- Parallel execution.
- Other advantages provided by the architecture.
 - Extensibility of functionality.
 - Ease of design.
 - Advantages in compiling.
 - Better control to the compiler, since the compiler is the most closest to the source code than the processor.
 - Instruction scheduling to take advantage of the fact that some values may only be available later.
 - Advantages over VLIW or similar architectures.
 - Advantages in looping and branching since these operations are schedulable.

5.1 Advantages in Fetching

[92] In device resulting from this invention, the instruction size will be smaller due to:

- The instructions can be compacted in Instruction Packet and therefore it will be able to occupy a lesser space than the word size of the PE.
- The removal of the op code will also result in saving of space.

5.2 Advantageous in Decoding

[93] Since the instructions are determinable by its position, the burden of decoding an instruction op code will not be placed on the processor will lead to:

- Reduction of power dissipated in decoding. This would also mean that power is saved in the implementation of complex pipe-line and parallel instruction execution mechanisms.
- Reduction of components (e.g. logic gates, transistors, etc.) used for decoding. This is the proximate cause of the power saving mentioned above. Moreover, these transistors can be used in other areas like to provide additional functionality and to boost processing power.

[94] All currently existing architectures use decoding. In VLIW and similar architectures, this is simplified but not eliminated as in this arrangement.

5.3 Advantages in Execution

[95] Since these registers are hard-wired when an Input Register is changed the Output Registers will change relatively faster than currently available processors. In the current context, the instruction will need decoding followed by identifying the registers involved, which of course consumes time and needs a lot of transistors to implement. The data in the operand register data will need to be sent to the processing unit and subsequent to the execution of the operation the result will need to be stores in the destination register. Identification of registers involved and the destination resisters will be eliminated in this processor design. This will save instruction time

and transistors and power dissipated in them. In addition only a "move instruction" is needed to execute an instruction.

[96] In VLIW technologies time and transistors (or any other equivalent) involved in making routing related decision with regard to the instructions, within the processor, may be saved in addition to the saving mentioned above. Moreover, a processor or a computing machine design using IISC will have advantages over VLIW based processor design, since the code bloat which results from excessive use of NOOP instructions will not manifest in code generated for an IISC processor.

5.4 Other Advantages

[97] This architecture has the advantage were more functionality can be added to a processor easily. The functionality addition can be across version of the processor as addition, which are dynamic if the processor is implemented on a FPGA or equivalent. Moreover, the design of an IISC based processor is simpler since there are only simple PEs which makeups the processor. The processor facilitates the execution of many instructions in parallel therefore; instructions can be scheduled by the compiler to better exploit these parallelisms within the programme.

[98] Across different versions of the chip, the number of certain units can be increased e.g. more mathematical Operation Mills/FU can be added for processors that are used in mathematical computations. This way the more widely used units can be increased for special purpose chips. The less frequently used units need not be increased. This facilitates more efficient use of the transistors on a wafer than introducing multiple cores on which where the whole chip is duplicated regardless on how intensively that part of the chip is in use.

[99] Smaller instruction size for a compiled programme (due to the lack of op codes) would result in less bandwidth needed for data transfer. In addition, the minimal use of NOOP will save bandwidth. This would be significant compared to pure VLIW architectures.

[100] There is an ability to schedule instructions, in practicing the art of the invention, will save the components and other resources used branch prediction. The branching condition in many cases will be known by the time of the execution. This will enable the instruction pipeline to be filled appropriately with minimal stalling due to I/O. In addition, since the cache is very manageable and the instructions executed are only from the cached, even if the condition is known late there will be minimal staling of the processor. (The condition should at least be computed in the previous execution cycle.)

[101] The invention facilitates better cache management capabilities for the compiler. Since the compiler is more closer to the Source Code, it will be able to make better decisions in terms of:

- What to cache,
- When to cache and
- When to discard from cache

Drawing:

6 Description of Embodiments

[102] The several embodiments described here are solely for the purpose of illustration. The various features described herein need not all be used together, and any one or more of these embodiments may be contained in a single embodiment as well as multiple embodiments may contain the features described as a single embodiment herein. Therefore, a person skilled in the art will recognise the other embodiments to be practiced; and modification and alterations that might be needed. The embodiments described herein contain the best possible mode and its variation contemplated of practicing the art of the invention as appearing to the inventor at the time of writing. With the change of the state of art in the future the embodiments and the practice of the art of the invention might vary while maintaining the essence of the invention.

[103] **Figure 1.: Main Architecture:**

[104] This picture depicts a diagram which is the Best Mode for the register and FU layout. The data bus is also depicted. The data bus moves values between the registers. The control unit of the data bus is called the Data Bus Controller for identification purposes. In this lay out a FU has multiple input and Output Registers. The bit patterns in the Input Registers are mapped into the Output Registers. The logic in the FU does the transformation. Generation certain output together may be easier so multiple outputs have an advantage. On the other hand, certain computation may produce multiple outputs.

[105] **Item 101.:** Multiple outputs are computed from the inputs, which are easy to compute together. Some of the outputs may be discarded. See calculations in the callouts above.

[106] **Item 102:** Example of a FU implementation.

[107] **Item 103:** Register data in directed using the register data bus.

[108] **Item 104:** Controls the flow of register data. Only instructions are to move data between registers.

[109] **Item 105:** Register flow control parameters (i.e., the operands of the implied instructions).

[110] **Item 106:** There are many FU which performer various functions.

[111] **Figure 2.: Alternate Architecture:**

[112] The alternate architecture is similar to the above but only differs from the fact that the there is only one Output Register for a FU.

[113] **Item 201:** Register data in directed using the register data buss.

[114] **Item 202:** Example of a FU implementation.

- [115] **Item 203:** Only one output is computed from a set of inputs.
- [116] **Item 204:** Only 1 Output Register.
- [117] **Item 205:** There are many FU which perform various functions.
- [118] **Item 206:** Controls the flow of register data.
- [119] **Item 207:** Register flow control parameters (i.e., the operands of the implied instructions).
- [120] **Figure 3.: Processor States:**
- [121] This depicts the states that a processor may go through. Three major possibilities are illustrated in the diagram.
- [122] **Figure 4.: Instructions:**
- [123] This diagram depicts the layout of instructions. The instructions are packed together to maximize parallelism giving heed to the availability of the FUs as well. If there were infinite FUs per given operation, then the only consideration that is needed in packing instructions is the interdependency between them. The number of FUs is a limiting factor in achieving the maximum possible parallelism.
- [124] **Item 401:** Selects a register. (This register's data will be routed in the next instruction.)
- [125] **Item 402:** Certain number of instructions that executed in parallel.
- [126] **Item 403:** Level 2 instructions are dependent on level 1.
- [127] **Item 404:** Routes the previously selected register to an input register.
- [128] **Item 405:** Dealing with Breaks in Dependency Levels
 The packet of instructions taken to the processor cannot be executed normally; since there is a break in the dependency level (instructions in a level are independent of each other). There are a few ways to overcome this.
- (1) Since there is a break in the level, the input data routed, subsequent to this is garbage if the whole packet is executed once. The work around is to execute the packet by the number of breaks that is there.
 - (2) Stall the processor until the results of the previous level is available.
 - (3) On retrieval (fetch) of the packet of instructions replace the non relevant instructions with dummy (NOOP) instructions. E.g. in the above case the second packet executed will have the last 5 instructions as NOOP, and the next instruction packed will have the first 9 instructions as NOOP and so on. Since fetching takes time on fetching the instructions this adjustment is made to the register holding the next executable instruction. The level boundaries should be specified in a previous instruction packed. The number level boundaries, which can be specified at once is equal to the maximum parallel instructions supported by the architecture. It should also be possible to queue more information about dependency levels than which can

be specified, in case there are as many dependency levels in a packet as the number of instructions.

[129] **Figure 5.: Alternate Instruction Layouts:**

[130] This diagram shows how instruction can be made schedulable. The first is a three instructions method which uses: select, target, set clock. The second diagram shows how a timer is set using a special FU.

[131] **Item 501:** This Diagram shows a variation of the instruction set where all instruction are schedulable.

[132] **Item 502:** Registers are output or input registers.

[133] **Item 503:** Both the diagrams represent the how operations can be made schedulable.

[134] **Item 504:** Registers are output or input registers.

[135] **Item 505:** This Diagram shows a scheduling can be achieved by moving a timer value into the FU's timer/scheduler.

[136] **Figure 6.: Register Routing:**

[137] Registers are moved around using the Data Bus. This is controlled by the Data Bus Controller, which takes the source and destination registers, as parameters and then transfers the registers values from the source to the destination. In fact the Data Bus Controller is what executes the instruction of the processor. As iterated, this is equivalent to a register select and register target, i.e., a single move instruction.

[138] **Item 601:** Assuming 5 parallel instructions are executed select 5 registers.

[139] **Item 602:** Assuming the output registers RO0 – RO6 are affected, due to the change in RI3, RI5, RI10, RI11, and RI14.

[140] **Item 603:** Copy register content to intermediate registers.

[141] **Item 604:** Copy register content from intermediate registers.

[142] **Figure 7.: Register Access:**

[143] This figure illustrated the proposed best mode for register access. It also illustrated the layout of registers. It further illustrates the Best Mode for moving registers, by coping them to intermediate registers. A FU may carry out a task for more than one cycle; in such a case the rising of the first cycle and the falling of the last cycle will be as illustrated.

[144] **Item 701:** The Written Bit of the input registers are wired to the Dirty Bit of the output registers. The Written Bit may activate a FU. On activation of a FU the dirty bit is updated to signal the Output Register is in the process of becoming overwritten. The dirty read interrupt would be raised by the Data Bus on access.

[145] The two overflow bits should not be used by a linked FU if the overflows need to be ignored.

[146] Some times the fact that a overflow occurred may be store in the other bits (marked as ...)

[147] All the bits of a register may not always be used. Some bits may be used as flags. The output register format should be recognized when moving it as an input register.

[148] **Item 702:** On write this bit is set and it will activate the FU(s) linked to this register. This is cleared in the next cycle.

[149] **Item 703:** The dirty bit is set when the input register have changed and the FU is still processing it. An interrupt may be raised on dirty reads. The dirty flag is cleared in the next cycle or when the output of the FU becomes available.

[150] **Item 704:** When an interrupt is raised (in the middle of a cycle) the currently accessed data is discarded and the interrupt is processed from the next clock cycle.

[151] The interrupt unit may function using the main clock cycle or a skewed (delayed) cycle or both.

[152] **Item 705:** This scheme saves on the connectors needed to move registers. The alternative is to provide connector from each register to every other register which is less than practical.

[153] **Item 706:** In rising of the cycle data is accessed. (Moved to hidden registers) The dirty bit is checked in this point.

[154] **Item 707:** Interrupt is raised if dirty.

[155] **Item 708:** In falling of the cycle data is written. (Written from hidden registers.) In this the dirty bit is updated.

[156] **Item 709:** Execute. Reset the Written and dirty bit.

[157] **Item 710:** The accessed registers are copied to hidden registers.

[158] **Item 711:** The hidden registers are copied to the relevant registers.

[159] **Item 712:** Move data from hidden registers to the appropriate registers.

[160] **Item 713:** Move data from hidden registers to the appropriate registers.

[161] **Item 714:** Move data to hidden registers.

[162] **Figure 8.: Code Level Management Unit:**

[163] Operations in an execution of programme have dependencies. This unit illustrates how dependencies are tackled if they happen in the same instruction packet. Information about dependencies is sent to the Code Level Management FU. This information is queued and used in fetching instructions to deduce whether a dependency boundary exists in any of the instructions that are going to be executed. If this is the case a special adjustment is made. There are many possible adjustments; the figure illustrates some of the methods that can be used. These methods include

insertion of NOOP instruction into the instruction pipeline upon fetching, stalling the processor on dependency break so that the next level executes in the next cycle, and executing all the instruction of the packet by the number of dependency level breaks in the packet.

[164] There is a possibility that this functionality is not needed if all the instructions are schedulable. Then dependency breaks occur the instruction will just be scheduled to run in a later clock cycle.

[165] **Item 801:** Code level management will be very important to maximize the throughput of the processor. This FU will deal with scheduling execution of code

(1) So there are no dirty inputs to FU,

(2) Manage potentially dependent instructions within an instruction packet,

The compiler will ensure that maximum leverage is gained by:

(1) Parallelism in operations,

(2) Availability of FU to carry out the operations.

[166] **Item 802:** 5 registers are ideal if instructions are executed in packets of 5 since there is a maximum of 5 levels per instruction.

[167] **Item 803:** R00 & R16 are both Input and output reg. for both Units.

[168] **Item 804:** Head of Queue -R00 (Next Level Break).

[169] **Item 805:** De queue when executed instruction packet has the dependency level boundary pointed by the first register.

[170] **Item 806:** Tail of Queue – R16 (Last Level Break Registered).

[171] **Item 807:** R00 & R16 are both Input and output reg. for both Units.

[172] **Item 808:** R00 & R16 are both Input and output reg. for both Units.

[173] **Item 809:** Instruction “splitting” can be achieved in 3 ways. They are listed in the order of complexity:

(1) Execute the packed repeatedly by the number of levels in it (In this case 2).

(2) When fetching show the instructions as 2 instructions with NOOP padding like in the left.

(3) Delay subsequent instruction dependencies until the needed results become available. (This might be too complex.)

[174] **Item 810:** Dependencies in packets are split.

[175] **Item 811:** First Instruction.

[176] **Item 812:** Second Instruction.

[177] **Item 813:** Instruction Packet is Split to Two.

[178] **Figure 9.: Data Bus:**

[179] The Data Bus is responsible for moving instructions from one register to another. The executable instructions contain parameters or operands to the Data Bus Controller unit.

[180] The Pre-fetched Instruction Pipeline holds the next instructions to execute. On pre-fetching certain adjustments are made to the instruction like dynamically inserting NOOP instructions. This is depicted in the figure for the code level management unit.

[181] **Item 901:** Move data between registers. Since some of the registers (input registers) are wired to FU input and others (output registers) are wired to the output of FU, all the computational tasks can be achieved through shuffling data between these registers.

[182] **Item 902:** NB: “Select register” and “target register” can be thought as two instructions or a single move instruction.

[183] **Item 903:** NB: A number of instructions equal to the number of instructions in a packet are placed in a pipeline. In this pipe line adjustments are made to account for multiple dependency levels. (A packet is at most split into different packets equal to the number of instructions in it.) The term pipe lines here are different from that of normal context where parts of the instructions which are independent are executed in overlapping manner.

[184] **Item 904:** Instruction Partition of Cache.

[185] **Figure 10.: Memory Controller:**

[186] The memory controller is an external dedicated PE. It moves memory between PE and the main memory.

[187] Memory bus should be wider than the input of the processor words or each PE may have its own Data Bus.

[188] **Item 1001:** The memory controller selects a block of memory and then transfers it to the cache of a PE.

[189] **Item 1002:** Instructions from the OS or PE to optimally manage the PE to maximize throughput.

[190] **Item 1003:** Instructions packet from the OS or PE, for system with 5 or less PE. PE # is the requesting PE # and need not be an input since it is implied. In this case a packet will only have (‘Memory Address’, “Number of Instructions”).

[191] **Item 1004:** A programme, PE or the OS managers the request to transfer data to a given PE. These requests may result from:

(1) A programme request certain instructions to be cached, as data or instructions. (PE can execute instructions which are cached only in a circular manner. This will reduce executable code size).

(2) OS assigning a task to the processor. In this case the instruction that needs to be executed along with the data is transferred into the processor by placing it in the cache.

[192] **Item 1005:** The catch management module will know where exactly (address and partition) to place the stream of instruction received from the memory.

[193] **Item 1006:** Memory

[194] **Figure 11.: Memory Management:**

[195] Memory management in this case is multi-phased. First, the needed memory elements are cached. Subsequently, this is accessed from the processor. The processor can only access the cache. The Memory Management FU, liaisons the movement of data from the cache to the registers of the PE and vice versa. The cache has two kinds of major partitions: one for memory and the other for instructions.

[196] **Item 1101:** Instructions from the OS or PE to optimally manage the PE to maximize throughput.

[197] **Item 1102:** Input Operands / control parameters.

[198] **Item 1103:** Both Cache Management and Memory Management FU would support conditionality of execution. MMFU deals with data held in memory which are loaded into the registers. DBFU deals with the operands of the instructions the processor will have to deal with.

[199] **Item 1104:** Input Operands.

[200] **Item 1105:** Input Operands.

[201] **Item 1106:** (*) Move instruction block. (*) Repeatedly move instruction block.

[202] **Item 1107:** Memory management instructions are as follows:

- (1) Move memory block to catch,
- (2) Move cached item to registers,
- (3) Move cached instruction block sequentially to data bus,
- (4) Repeatedly move cached block in sequential manner to Data Bus FU.

[203] **Item 1108:** Multiple PE or processors.

[204] **Item 1109:** The memory management instructions can also be executed using a Memory Controller FU just like any other instruction. (E.g. transfer a memory element in cache to registers.)

... M – Memory element

... C – Cached element

... R – Register

[205] **Figure 12.: Memory Controller in Machines with Parallel PE:**

[206] In scenarios with parallel processors, each PE may have its memory pool. This serves as a local cache.

[207] **Figure 13.: Explicit Cache Management:**

[208] The compiler is the closest to the code and is in the position to better optimize the variables and instructions to cache for faster access. This processor architecture addresses this issue by facilitating greater control of the cache by providing ways and means to explicitly control the cache through FU. This arrangement gives the compiler also more control.

[209] **Item 1301:** Instruction is placed in a special partition in the cache designed for it. When the boundary of the instruction cache is reached then the Instruction Pointer (IP) is zeroed. Instructions are loaded in circular fashion into the catch. (I.e., if a transferred block goes beyond the instruction boundary then it is placed at the beginning.)

[210] **Item 1302:** The condition is used in conjunction with conditional branching across a large number of instructions which is not in the catch. (If the jump is within the instruction of the cache, then a simple IP adjustment will suffice.)

[211] **Item 1303:** NB: Parts of cache will be stacked across function calls.

[212] **Figure 14.: Looping & Branching:**

[213] Looping a branching constructs are very important in any processor. This architecture has a novel approach to both looping and branching. Looping and branching is schedulable, i.e., the looping and branching statement can be sent to the PE before hand. The instruction will trigger when the scheduled number of clock cycles have elapsed.

[214] **Item 1401:** Branching will be carried out by a branch scheduling FU. This would instruct to reset the IP to a specific location in the cache, after a given number of clock cycles or executing a specific number of instructions. (PE cannot directly address memory.) The test for the jump condition should be done early as possible to ascertain which direction the execution would flow.

NB: the PE implementation most probably will not stall; therefore the compiler must handle possible delays in loading instructions.

The start latency can be used to carry out some initialization instructions. On the other hand the loop or jump can be pre scheduled using this feature.

Different latencies are used to make sure that special tasks like loading a different instruction block, can be done depending on the condition (such tasks may take different time).

Packets may be scheduled before hand to ensure that the whole packet or perhaps the whole pipeline (or major part of the instruction pipeline) gets executed.

[215] **Figure 15.: Stack and Call Management:**

[216] This diagram shows how the stack is managed. On function call the cache window is switched. The switch is such that some of the parts of the cache will overlap with the previous window. The overlapping portion can be used to pass and return parameters.

[217] In the case of register there would be many shadow registers. These shadow register are access throw the window associated with the context. On change of PE context, i.e., on function call; the window on the registers changes. That is a different set of registers will become the active registers.

When the stack get full the register are spilled to memory and as it becomes empty if there are spilled elements they are loaded. The stack for this purpose should be circular.

The stack management FU can be used in some implementations to change the process context. This scheme is discussed above.

[218] **Item 1501:** Registers are parts of the PE context and will also be stacked. For simplicity they will not overlap.

[219] **Item 1502:** On function call stack the PE context.

[220] **Item 1503:** The stack element beyond the top is wired to the current PE context so it is updated without latency.

[221] **Item 1504:** Common cache area. (Saved on change of process context).

[222] **Item 1505:** Cache area which overlaps between function calls. (Saved on change of process context.).

[223] **Item 1506:** If queue is more than the threshold spill the registers into the memory stack and increment lowest PE context. If it is lower than a threshold and there are stacked PE contexts load them after the lowest PE context and decrement the pointer. (This is to accommodate function calls or returns which are in quick succession.)

[224] **Item 1507:** Last PE Context Stacked (Top of Stack).

[225] **Item 1508:** Lowest PE Context (Bottom of Stack).

[226] **Item 1509:** NB; in this processor the cache is also an integral part of the PE context. Therefore, it also needs to be stacked.

[227] The addressable internal cache size should be small to enable fast operations.

[228] **Item 1510:** NB: The cache will have a part which will not be saved as a part of the context on function calls but would be saved on change of process context. Needed global variables will be cached as needed and will be stored here.

[229] In addition, there are overlapping areas. This can be used for passing and returning of variables.

[230] **Item 1511:** NB: Unlike traditionally a process will not enjoy a separate stack. A Stack would be for a PE regardless of the number of processors run by it. The heap changes from process to process. (Use of Automatic variables is very much

encouraged to avoid memory fragmentation. The compiler will try at best effort to convert dynamic variables to automatic with the best effort. E.g. a class variable or global variable which is only accessed in one function can be made an automatic variable.)

[231] When the process context needs to be changed the PE will be interrupted. This will cause the PE context to be saved. It might lead to a spill which would be put to the general stack. After a series of switchers the PE context will be switched back to the previous by popping the stacking. On popping the stack the stack state will be spilled to memory if the popped process is still active. It would also be possible to pop all values at once. The OS will decide where to store this data and will be stacked when needed.

[232] **Item 1512:** Spill to Memory gets from Memory.

[233] **Item 1513:** Interface with Memory.

[234] **Item 1514:** Common and overlapping areas in the cache are saved.

[235] **Item 1515:** Restore common and overlapping areas in cache.

[236] **Item 1516:** After a certain number of process context changers and when the process stack gets full. The processors will have to switch back to previous.

[237] **Item 1517:** Process stack.

[238] **Item 1518:** Process stack bottom.

[239] **Item 1519:** The process stack will be used to stack the common cache areas and overlapping areas in the cache.

[240] **Item 1520:** Free Memory.

[241] **Figure 16.: Data Cache Window:**

[242] The data cache partition is windowed, i.e., the whole cache cannot be accessed. The access to the cache is through this window.

[243] **Figure 17.: Instruction Fetching FU:**

[244] The Instruction Fetching FU draws a stream of instruction from the instruction partition of the cache. As iterated above necessary adjustments are made to the instruction packets.

[245] **Item 1701:** The instruction fetching FU transfers data from the cache and pass it on to the data bus to be transported to the relevant FU for execution. In this process certain adjustments are made.

[246] **Figure 18.: Process Management:**

[247] The process management FU can be implemented two way. This diagram is suitable representation of both.

[248] The process context switching is described above.

[249] **Item 1801:** The PE context across process will be saved as a combination of two stacks. The main internal stack is to handle function calls, so it has common and overlapping areas (out of two which needs saving on change of process context.) These parts on the main stack will be stacked in a secondary stack. The size of the secondary stack determines the number of process contexts that can be switched without switching back. (This secondary stack is not spilled into memory. Also, if the main stack does not have common or overlapping areas this secondary stack will not be needed.) Two interrupts will handle the switching between processors and switching back.

[250] **Item 1802:** Memory accessed on switch back of process context. The previous process context will need to be immediately committed to memory. This will cause the processor to stall until the commit is complete. Since the context is dynamic this instruction cannot be scheduled.

[251] **Item 1803:** Output registers of the Process Management FU is mapped or wired to the input registers of the Stack and Call Management FU.

[252] **Figure 19.: Process Management - An OS Perspective:**

[253] At the OS level the process is managed using a process context table. The table stores the relevant data needed by the OS. In the case where the stack is used for processor context switching, when the process context is switched back the previous context will store in this table.

[254] **Item 1901:** When the process is switched back the stacked data is spilled to a location in memory. This saving can be scheduled.

[255] **Figure 20.: Alternate Process Management:**

[256] In this scenario the process management data is stored in a table. The process data is moved to and from the table. In this case the stack data will also need replication. In the previous design memory elements are conserved but it is complex to implement. The stack also will need to be spilled many times into memory when switching back.

[257] **Item 2001:** Memory access is needed when table is full. This needs to be handled by the compiler. Since a delay is tolerable this instruction need not be immediate, i.e., it can be scheduled.

[258] **Item 2002:** An alternate implementation is to have the process table implemented in the processor it self. Process context is switched by placing the registers and cache in one of these slots. Any process can be switched in and out.

[259] **Item 2003:** Memory access is needed when queue is full. This can be transparent to the compiler and process. Since a delay is tolerable this instruction need not be immediate, i.e., it can be scheduled. This might be the least time consuming implementation.

[260] **Item 2004:** In the queued technique the process context are queued. When the queue becomes full new process context entries are spilled to memory. (This is transparent to the process and compiler). The context can be switched back only by de-queuing.

[261] **Figure 21.: Processor Initiations:**

[262] In the processor initialization a programme from the ROM is loaded to the cache of the processor. The execution begins from the first instruction in the instruction partition. These instructions are sent to the Data Bus. These instruction loads additional FU needed for the operations of the processor. New connections between registers and FUs are made during this phase.

[263] **Item 2101:** This processor ideally may not have explicit memory addressing. Memory blocks can be requested to be cached and this cache can then be accessed by a PE. On boot up the initialization data and instructions in transferred to the processor. A PE will have some basic FU which are hardwired to certain registers. Subsequently new FU and are loaded and new connections are made between FU and registers. The registers may be flagged as read only. The loading of FU and connections are made using a special FU. The special FU needed for this initialization should be prefabricated and hard wired FUs.

[264] The process loading is also done in a similar manner.

[265] **Item 2102:** Memory access is needed when queue is full. This can be transparent to the compiler and process. Since a delay is tolerable this instruction need not be immediate, i.e., it can be scheduled. This might be the least time consuming implementation.

[266] **Figure 22.: PE Initialization:**

[267] In the PE initiations process FUs will need to be loaded. This will be from external sources. I/O system will be used to load the FUs. In this process connection are also made, between the registers and the FUs. Special FUs is used for this process. The functional units that are used in the initialization process should be pre-fabricated and the interconnections should be pre-established.

[268] **Item 2201:** FU are loaded from a well known device. The parameters will include all the needed information from where to load the data and the number of bytes to transfer etc.

[269] **Item 2202:** The FU will have a certain number of connection leads which can be used for input and output. They may not all be used.

[270] **Figure 23.: Interrupt FU:**

[271] The interrupt FU interrupts the processor in the task it is doing to attend to another task. The interrupts are stored in the instruction partition of the cache. The code size of an interrupt should be minimal therefore all the code may not be cached. These stored interrupts may load from memory any additional code it might need. Some of the interrupts which are needed during initialization as well as a few other main interrupts may exist in ROM and will not be changeable to the executing programmes. Interrupts can be two types: Hardware and software. In the case of

hardware interrupts the Interrupt FU is signalled by sending a signal from the Hardware Interrupt Controller to indicate whether there is a problem.

[272] **Item 2301:** The interrupts are stored in a section devoted for it. Some of the code may be read only. In any case the last of the interrupt points to a section to execute.

[273] **Item 2302:** The hardware FU signals the Interrupt unit while loading the parameters to well known registers.

[274] **Item 2303:** Interrupt is invoked by loading the registers.

[275] **Item 2304:** The interrupt FU handles software and hardware interrupts. Interrupt FU maintains connections with the threading FU. In a blocking interrupt the threading context is switched.

[276] **Figure 24.: Thread Scheduling:**

[277] Threading is a significant development in the computing scenario in recent times. A list of active threads are maintained in the cache. The start of threads and the end of threads are maintained. If needed parts of the threading table is spilled into memory. Such an activity should be handled by software.

[278] When the thread is spawned it might not be in the cache. In such a case the memory location of the instructions and the number of instructions to cache and the cache location can be specified. In the case that the thread is already in the cache the cache location can be specified. In both cases the instruction can be scheduled.

[279] **Item 2401:** Thread context is maintained in the active thread table. On a blocking interrupt the active thread is switched to another.

[280] **Item 2402:** Only used when it is needed to get from memory. One block in the thread table.

[281] **Item 2403:** Active Threads is moved in circular manner.

[282] **Figure 25.: Time Slicing FU:**

[283] Time slicing is used in to run many processors as if running simultaneously. This diagram illustrates the basic FU and register layout needed.

[284] **Item 2501:** NB: In the stack based process context switching the Time Slicing FU interacts with the stack.

[285] **Item 2502:** Output registers of the Time Slicing FU is wired to the Process Management FU.

[286] **Item 2503:** In the case of where the internal stack is used for process management.

[287] **Figure 26.: BNF to Intermediate:**

[288] In compiling the programme is parsed. This parsed tree is then converted to a graph theory based model. This graph is used to determine dependencies. This is further transformed considering the availability of FU as a limiting factor.

[289] **Item 2601:** BNF will be mapped into a graphical representation highlighting which models operations as vertices and dependency between operations as edges. Dependencies arise due to the flow of data from one operation to the other, i.e., the results of certain operations are needed for the other operations. Effectively a dependency graph is a graph showing the flow of information from operation to operation. The operations transform the data as it flows through.

[290] **Item 2602:**

1) BNF data is converted into a Dependency Graph.
2) The Dependency Graph is then further converted into a Slotted Dependency Graph (a Dependency Graph which is adjusted for the availability of FU.) In this graph the vertices will represent operation and the edges will represent dependencies. In addition, there will be a number of slots which an operation fits into. These slots represent the available FUs in a given clock cycle.

[291] **Item 2603:** Convert to FU centric graphical layout.

[292] **Item 2604:** Convert to FU centric graphical layout.

[293] **Figure 27.: Code Arrangement:**

[294] Depending on the availability of FUs the dependency graph will need further transformation. In a given level there can only be instructions that are independent and the instructions present should also match the type and number of FUs available.

[295] **Item 2701:** Assuming that all the operations use the same FU and there are only two FUs for the operation, the above dependency graph will be transformed to the given Slotted Dependency Graph.

[296] **Item 2702:** This diagram shows dependencies among operations. In addition to the dependencies among operations there are two limiting factors which are

- (1) The number of FU for a given operation,
- (2) The instruction packet size.

[297] The compiler should analyze the dependency graph and the availability of resources (FU) and optimally arrange the instructions in packets. Instructions in a packet are executed in parallel.

[298] The instructions are arranged in such a way the operations which many other operations depend on, are executed in advance, in order to achieve maximum throughput.

[299] **Figure 28.: Compiler Extensibility:**

[300] This figure further elaborates code arrangement and matching process. Since the FUs can change, the functionality of the operations the processor can perform should be determined.

[301] **Item 2801:** Since the FU is changing, the compiler should be specified in a language which defines how BNF is converted into a FU centric representation. Each

FU should have a compiler understandable description and how to resolve conflicts if there are more than one FU competing a certain pattern in the parse tree.

[302] **Item 2802:** Dependency graph based on atomic operations.

[303] **Item 2803:** Match pattern in the parse tree with patterns for a FU.

[304] **Item 2804:** FU centric dependency graph.

[305] **Item 2805:** FU description using a language which defines the logic of the FU.

[306] **Figure 29.: Compiling High Level Language (HLL) to Logic:**

[307] This diagram depicts how FUs are to be programmed. HLL constructs are directly translated to logic.

[308] **Item 2901:** A big problem faced by programmers in designing hardware is to think of terms on logic circuits. To overcome this difficulty HLL code can be translated into logic and hardware design opposed to machine code to be executed on a processor. This technology can be used in designing FU.

[309] **Figure 30.: Array Registers:**

[310] Two approaches in accessing Very Long Registers are specified here. In one scheme virtual register are used as a window to part of the long register. In the other scheme a special FU is used to write and read parts of the long registers.

[311] **Item 3001:** Generally registers are word sized or less. Very long bit register can be accessed using a special FU as shown here.

[312] **Item 3002:** Moving memory from the cache to a long register and vis-à-Vis will be similar to moving data for normal registers. This should be done by the Memory Management FU in one step.

[313] **Item 3003:** Very Long Bit registers can be accessed as array of normal word sized registers. A drawback of this approach is that the number of registers will significantly increase.

[314] **Item 3004:** Virtual word sized registers which represent part of the underlying long register.

[315] **Item 3005:** Moving memory from the cache to a long register and vis-à-Vis will be similar to moving data for normal registers. This should be done by the Memory Management FU in one step.

[316] **Figure 31.,: Register FU Connectors - using mux:**

[317] **Item 3101:** The multiplexers are connected to a certain number of registers.

[318] **Figure 32.,: Register FU Connections - using mux & demux:**

- [319] **Item 3201:** The multiplexes are connected to a certain number of registers.
- [320] **Item 3202:** All multiplexes and de-multiplexes receive a select input from stored memory elements.
- [321] **Figure 33.,: Register FU Connections - using demux & mux:**
- [322] The figures 31 ... 33 show some of the modes contemplated in establishing the connections between registers and FUs.
- [323] **Item 3301:** The multiplexes are connected to a certain number of registers.
- [324] **Item 3302:** All multiplexes and de-multiplexes receive a select input from stored memory elements.
- [325] **Figure 34.,: Execution Cycle:**
- [326] This figure depicts the execution active and passive instructions and their properties. Active operations are the 'read' and 'write' operation. Passive operations are the activities carried by the FUs. Active operations in a packet are executed in parallel and possibly many packets are executed in the space of a passive operation. If this is the case all the packets executed seem to be parallel to the programme since all the instruction has done is move certain values in registers to input registers which are only accessed at the beginning of a passive operation.
- [327] **Item 3401:** Execution of 4 operations in parallel. This implies that the instruction packet size is 4.
- [328] **Item 3402:** Time Taken Execute a Passive Operation by a FU.
- [329] **Item 3403:** Possibly in the same instruction packet.
- [330] **Item 3404:** Active operation Execution Time.
- [331] **Figure 35.,: Ideal Execution Cycle:**
- [332] This figure depicts a more idealistic approach where the 'write' of a previous 'read' are carried out in the same time as a fresh 'read'.
- [333] **Item 3501:** Execution of 4 Read and write operations in parallel.
- [334] **Item 3502:** Parallel write of a previously read value with a fresh read.
- [335] **Item 3503:** Quasi parallel execution.
- [336] **Item 3504:** Active operation Execution Time.
- [337] **Item 3505:** Parallel Operations.
- [338] **Item 3506:** Time Taken Execute a Passive Operation by a FU.
- [339] **Figure 36.,: Long Registers:**

[340] Long registers are registers used to process large chunks of data including but not limited to vectors and matrices, strings, arrays, bit strings. These registers can be used along with FUs to process the whole chunk of data in one operation.

[341] **Item 3601:** Array Registers / Bit String Registers / Byte Buffer Registers / Very Long Registers.

[342] **Item 3602:** Window.

[343] **Item 3603:** Overlapping Window.

[344] **Item 3604:** Normal Register Windows.

[345] **Item 3605:** Window position.

[346] **Item 3606:** Value.

[347] **Item 3607:** Array Registers / Bit String Registers / Byte Buffer Registers / Very Long Registers.

[348] **Figure 37.,: FU Configuration:**

[349] Operations of a FU can be altered by loading a new FU to occupy its slot or by providing configuration to alter the operation carried out. This shows how to provide configuration details so that the operation of the FU can be altered. The configuration can be in a separate register or part of a register. The latter method is more advisable when using long registers.

[350] **Item 3701:** Long Register.

[351] **Figure 38.,: Switch:**

[352] The 'switch' statement is used in many programming languages. The diagram shows how the switch statement can be supported by a FU. The value is put into the relevant register and compared with the 'cases' which reside in other registers. If there is no match then the execution flow continues normally. Else a branch will occur to a given destination at the given time. If the time is not set it would mean the jump is immediate.

[353] If there are more than the numbers of 'case' values supported by the FU then these 'case' values can be tested in manageable subsets. If there are less value repeated values can be used. The values are tested in a precedence order of registers. Therefore the first match will execute if there are repeats.

[354] **Item 3801:** If the value does not match the control will pass to the next statements. These statements can readjust the case values in case there are more than 4 cases. In case there are less than 4 case values the extra registers can be set to a value of a previous case. If there are two case values which are equal then the first is executed.

Best Mode Disclosure:

7 Best Mode

[355] In building a PE there are many decisions to be taken. The best mode contemplated in practising the art of the invention, as appearing to the inventor at the time of writing, is presented herewith.

7.1 Ideal Multiplicity of Instructions

[356] This invention should ideally have a move instruction, which can be broken to two smaller intuitions. One instruction selects registers and copies them to an array of hidden registers. The next instruction copies this from these registers to destination registers. This scheme will even enable a register swap within the same execution cycle.

7.2 Connections between the Registers and FUs

[357] With regard to the input and Output Registers, the best would be to be able to have more than one Output Register. Depending on the circuitry needed to connect the register there could be a restriction on the number and the registers which can be connected to a FU. Moreover, some register could be permanently connected to FU. These considerations will save on the complexity and number of the logical constructs needed create a PE. In case speed is paramount all the FU and the connection with registers can be pre fabricated, in such arrangement the flexibility is compromised for a gain in speed.

7.3 Switching between Processors

[358] In devising a scheme to process switching it would be ideal to choose the mode where there are process tables “internally”, rather than the stack based approach. The later saves on memory elements internally but it do need more memory accesses thus consuming more memory I/O bandwidth. This would pose as a bottleneck. Therefore, the scheme described first (shown in figure labelled: Alternate Process Management) is the ideal scheme, and would constitute the best mode of practicing the art of the invention.

7.4 Operation/FU Activation Scheduling and Dependency Level Breaks

[359] In practicing the art of the invention, if all the FUs were schedulable then many of the issues regarding “dependency level breaks” will not arise. Operations (carried out by FUs) in an Instruction Packet, which is not in the same level can schedule it self to be executed in the next clock cycle. However, in doing so the FU which executes an operation will not be free until such time the instruction is fully executed. Therefore, many FUs of the same type will be needed to ensure smooth execution. In addition, setting up the timers too will consume instructions (executed by the Data Bus Controller) or memory (the space occupied by the recurring operand used to set the clock, of some, which might not be even needed). Having the ability to schedule operations as well as manage Dependency Levels by padding with NOOP, would be the best mode of practicing the art of the invention. Timing/Schedulable operations can be used for operation of which require branching, I/O, and are power

consuming (if power consumption is an issue). By default the timing register could be zero and will remain so until a value is moved to it, i.e., the instruction will execute immediately. A FU to set all or a set of timer registers to a given value would be advantageous. In case an output register can be connected to two FUs, the use of times would be needed to ensure that the data in the output registers do not get write to mutually thus getting corrupt.

[360] If all instructions are schedulable, there is a possibility that an instruction will have:

- The register is selected for moving,
- The targeted register is selected for the data to be placed, and
- Time in which to activate the FU.

This scheme will not use more clock cycle for scheduled activation of FU. Nevertheless, a lot of space will be wasted for the timing details. Therefore, the timing details can be an Input Register to a FU. This will have some clock cycle (time) cost in setting up the registers but memory as well as memory I/O bandwidth will be saved. The clock register may not be a full length register. In such a case, when coping to these Timer Registers some higher order bits may get truncated. Using a scheme to Manage Dependencies as discussed would be less time-consuming owing to the fact that it takes only an instruction to register a Dependency Level Break opposed to multiple instructions for setting up the timers.

[361] Using timing and scheduling would be beneficial. Its use will also save power. Having the ability to specify Dependency Levels Also would be beneficial.

7.5 Ideal Number of Instructions

[362] Ideally, there should be a 'move' instruction only. Any time setting scheme should be handled by moving data to the FUs timer/scheduler Input Register.

7.6 Clock Cycle to be used by a FU in Carrying-out a Computational Task

[363] There is a possibility that FUs may consume different number of clock cycles. For simplicity for compiler design with regard to scheduling of instructions and managing of dependency level breaks it would be ideal if an instruction would only take a fixed number of cycles to execute.

7.7 Use of Packets

[364] Instruction of the PE can be moved in as packets. Instruction within a packet will be executed in parallel and more than one packet may be executed before a passive operation by a FU. In this case all the instructions executed within the time will be quasi parallel since from the programmes perspective the smallest quantum of time is what is taken for a passive execution.

[365] Using packets more instruction can be executed than using a stream of instructions so this would be the best mode of practicing the art of the invention.

7.8 Activation of FU

[366] The best mode contemplated for the activation mechanism of a FU is as follows. The activation of a FU will happen on writing to any of the input registers.

After the output is calculated the FUs becomes in active again but the values of the Input Registers and Out Put Registers will be maintained until such time that they are overridden.

[367] Other than these variations, the above-described method is the best way to implement the PE.

7.9 Dirty Reads

[368] Ideally dirty reads and it prevention should be handled by the compiler, thus freeing the need for hardware support to track it. Some times the ability to read a value in an Output Register just before it is updated in the next cycle may result in more optimised code too.

7.10 Very Long Registers

[369] If very long registers exist they should be manipulated using a FU. Using virtual register mapping would greatly increase the number of registers thus increasing the number and complexity of connection paths.

7.11 Caching

[370] The caching method in this invention may be used in conjunction with the caching schemes available today. An out level cache may ideally use the Harvard Architecture.

7.12 Timer/Scheduler Values

[371] When packet size changers across different series of the same processor architectures so would the clock cycles for an execution of a times operation. Therefore, the scheduler should be in terms of the number of active operations that should laps to trigger execution.

7.13 Parallel Execution

[372] The active operations which result from the execution of instructions should happen in parallel. The write of a previous read and a fresh read should be carried out simultaneously.

7.14 Scheduled Instructions

[373] If all the instructions are schedulable then the need to manage dependency levels will be significantly reduced. Therefore, ideally all instruction should be schedulable.

8 Mode for Invention

[374] This processor can be implemented on a silicon wafer as with many other contemporary processor implementations, but this does not need to be restricted to this form of implementation. If a novel, way to fabricated logical components is found this can be ported to such a scheme. This mode of practice is present in the prior related arts and will not pose as a problem to a person skilled in the art.

[375] The manner, in which art of the invention is practiced, as contemplated herein, may be varied and changed to suit, by a person skilled in the art. The state of art is a

rapid flux of change, therefore a person skilled in the art may change or adopt the manner in which the art of the invention is practiced, whilst keeping to the spirit and scope of the invention, as the state of art evolves.

9 Industrial Applicability

[376] This kind of PE can be used to carry out any computational task currently done using computing machinery. The FUs will become simpler to design and make in practicing the art of this invention. FUs are embodiments, which are already found in the prior related arts and can be readily created by a person skilled in the art.

[377] Many of the FUs can be optional; therefore, different PE can be implemented according to need. PE for super computing and scientific computing can be designed with all the mathematical and vector processing functionality needed. Moreover, simple processors for mobile devices and possibly simpler devices can be made using PE of this kind. This is due to the simplicity of design and the embodiments.

[378] This invention will have numerous advantageous effects. These advantageous effects will make this invention more viable than the solutions found in the prior art. The benefits reaped, in practicing the art as contemplated herein, will present itself in monetary and non-monetary terms due to the performance increase and advantageous effects it would create in practicing certain elements in the art of computer programming, and art of manufacturing computing machinery and apparatus; and in practicing their related arts and crafts.

Claims of the Invention:

10 Claims

What is claimed are methods and modes of practice as presented below. The following is only for illustrative purposes only thus this invention can be practiced in alternate forms and modes or using selective forms and modes, while maintaining the scope and spirit of the art of the invention. A person skilled in the art would be in the position identify such modes of practice.

[1] The crux of this invention is that there are Functional Units (FUs) (embodied apparatuses which carry out a computational operation or function) which has registers (embodied apparatuses which can be used as memory elements) that are hardwired to FUs. The inputs of the FU assigned to the input registers which are wired to the inputs of the FU, on computing the output the results are then written back into output registers which are wired to the output leads of a FU. Not all the registers in the system may be connected to a FU. Some registers may be dedicatedly connected to a FU where as some registers could be dynamically re-assigned to FU using a special technique. These connections may be crafted as a combination of multiplexes or as seen fit by a person skilled in the art. The FUs discussed here may be dynamically loaded or prefabricated such that it cannot be changed. If they are dynamically loadable, they are loaded and unloaded in the course of execution using special FUs; or they could be loaded by a pre-usage configuration mechanism or a dynamic configuration mechanism which uses a controller which among other things loads and unloads the FUs dynamically. The instruction for the controller may be issues by a separate programme or by the programme executed by the FU it self. In addition, the registers discussed here can be word sized registers or multiword sized registers which can be used to represent vectors, bit strings, strings, arrays, etc. A person skilled in the art will be in the position to recognize the optimal size, arrangement, and how to implement any special representations of data, in relation to the context of use. The registers can connect to one or more output leads or input leads of FUs or both. In the case of FUs, they may produce one or more outputs for a given set of inputs. The outputs produced collectively would include but not limited to, intermediate values of a computation which is likely to be re-used else where, or values which are easily deducible when producing one of the outputs, or values which are incidentally produced during a calculation (the inverted bit representation), or computation which logically produce more that one value (e.g. calculating a set of coordinates). A person skilled in the art shall recognize how to practice this aspect of the invention. Apart from producing output values, some FUs may be used to cause other side effects (other than producing an output value though this itself is a side effect) including but not limited to, the altering of the computational apparatus's internal state or configurations. Also, certain input registers may be containing configuration parameters for the FUs or the computing apparatus as a whole. The latter case can be implemented by special FUs which take in the parameters in stipulated registers and do the alteration of the computing apparatuses configuration.

[2] The special arrangement of the connections between the embodied apparatuses – namely registers and FUs – as in claim [1] above where the registers are hardwired to FUs, would facilitate an Instruction Set Architecture (ISA) of the processor that can

be implemented as a set of multiple recurring instructions (Multiple Recurring Instruction Set Computing); thus making the instruction deducible by its mere position in the instruction stream. Since the instructions are deducible by position the instructions are implied, therefore, an op-code is not needed to identify them. The instructions need not be decode too. This facilitated the mode of computation which do not require an explicit instructions for carrying out a computational task, i.e., though there will be instructions, the instructions themselves would be implied. Only the operands of the instructions will be provided and some of provided operands may be shared by more than one instruction.

[3] The mode of computation, as in claim [2] above, where the instruction are arranged in multiple recurring fashion, would facilitate ISAs which has one move instruction (computing using a single instruction or Single Instruction Set Computing – SISC), a select register instruction and target register instruction (computing using two instructions or Dual Instruction Set Computing – DISC) or other sets of instructions as identified by a person skilled in the art. The instructions in the instruction set would be in recurring blocks of instructions in which each individual instruction would appear in pre defined order.

[4] The mode of practice as in claim [3] above wherein would have further application as contemplated. This method among many other application, many be used as a scheme to reduce instruction size. Also in this scheme, dummy instruction can be accommodated by using special operand codes. When an instruction relates to moving register data, using special operand values, is analogous to using dummy virtual registers as the operands.

[5] The mode of computing as in the claims [1] and [2] where registers are connected to FUs, would include topologies there FUs are pre fabricated as well as FUs which can be changed or a combination of the two. Further, the FUs may have prefabricated connections between registers and FUs or connections which can be changed or a combination of the two. Generally the ability or combination possible connections as well as the size and/or arrangement of the FUs can be restricted hardware simplification. The changers in connections and loading and un-loading or FUs can instrumented using other specialised FUs or perhaps a special configuration controlling apparatus. In the first case the re-configuration will be plausible during the course of execution a programme.

[6] The dynamic connection as in claim [5] above can be implemented using multiplexes and de-multiplexes or similar devices. The arrangement can be such that the output node (source node) can be coupled to a virtual node using a multiplexer or a de-multiplexer. The connection the virtual node to the destination can also can be instrumented using either a multiplexer or de-multiplexer. The connections can also be implemented with a set of multiplexes associated with each output lead and register so that it can connect both source and destination. This can be done also using de-multiplexes in association with the sources, i.e., input leads and registers.

[7] The topology and mode of computation in claim [1] above and claim [2] above where the FUs are independent embodiments, would facilitate a scalable ISA where functionality can be added and removed as needed and with minimal effect of the rest

of the processor. When this is further coupled with the dynamic reconfiguration arrangement as in claim [5] above, a dynamically changing ISA will become possible.

[8] The embodies apparatuses in the form of FUs as in claim [1] above where FUs carry out computational tasks, could be activated on events including but not limited to the following:

1. On updating any of the input registers or on updating a selected set of registers or on updating of a sub set of a selected set a selected set of input registers.
2. On reading a value of an output register or a set of output registers of a given FU.
3. On external control events.
4. On control events generated by other FUs.
5. On scheduled timing events.

The use of these activation mechanisms can be used to preserve and output until next event. This includes but is not limited to:

- The next computation will only be carried out on reading the output.
- The next output will only be produced after a certain laps of times.
- On updating an input register of a set of them will produce an output.

Scheduled timing events include scheduling of FU operations which can be carried out by placing the schedule as a parameter in and input register. Other forms of scheduling can be done by the controller or other FUs. There also could be special FUs which can carry out the scheduling. Moreover, scheduling can be used to implement timed branching and loops. FUs can be scheduled to produce output or its side effects in multiple schedules of produce different outputs and side effects in different schedules. The latter will include but is not limited to, having different schedules for different branching operations.

[9] The method of claim [8] above, where the FUs are activated upon events including but not limited to reading of output, writing of input, a determinable time been reached – can be used to power saving and a heat reduction scheme.

[10] The method in item 2 of claim [8] above where the FU carry out operation upon events, including but not limited to, FUs which address memory in such a way that upon reading the value referenced by the FU, it would then point to the next memory address or an addresses logically inferred from the current state of memory references.

[11] The mode of computing as in the claims [1] and [2] where FUs operate on input data provided input registers to produce output data which is placed in output registers; would comprise of active and passive operations, i.e., this invention will facilitate carrying out an a computation using as a side effect of an active operation. The active operations result from execution of the instructions and passive operation which result as side effects of the actively carried out operation. There will be two sets of apparatuses which carry out the active operations and the ones that carry out the passive operations.

[12] The mode of practice in claim [11] above, where FUs have a set of input and output register where placing values in the former would result in the output been available in the latter would include. the active operations, among other operation, will include an explicit or implicit move instruction. When data is move to other

registers which are hardwired to FUs the output of it will automatically become available in output registers; this is a passive operation. All or most of the programme specific operations will be executed as passive operations.

[13] Moreover, the feature in claim [11] above would include, active and passive operations may execute at the same speed or alternatively may execute at different paces. If the execution of active operations is many folds faster, this can be used to parallelise many passive operations. In case the number of parallel passive operations which need to be executed is less than the number facilitated by the computing apparatus, the processing of active operations or class of operations can be stalled using a special FU.

[14] The feature of claim [11] and [12] would include active operations described here, would contain a scheme to move data from registers to other registers. This could be using intermediate registers, main memory, and connection paths. These movements of data – among other events including external events signalled by the controller, and operation including operation in the controller and its interventions and interactions and internal processing – cause the desired side effects of the passive operations. The example presented here is solely for illustration purposes, a person skilled in the art shall recognise any alternate application of this principle within the spirit and scope of the invention.

[15] The embodiments in claim [1] above where registers can be in the form of multiword sized registers or where the size of the some registers are multiple of regular registers, could be implemented as a collective addressing of a group of regular registers or as specialised multiword registers which can addresses as a collection virtual registers of regular size. In the latter case, the registers will become addressable through a virtual register scheme. The virtual registers can act like any other registers and may for input or output connections with FUs.

[16] The embodiments in the form of registers in claim [1] above where there are input and output registers, could include dedicated set of registers for input and/or another dedicated set registers for outputs and possibly other registers which do not participate as input or output registers.

[17] The scheme in claim [2] where decoding and instruction is not necessary can be used as a scheme to improve speed.

[18] The mode of practice in claim [11] above where there are active and passive operations can be used as a scheme to carryout as many programme related operation by executing many active instructions in parallel. The number of parallel instruction executed can be different in variation of the apparatus. If the number of parallel instructions taken at once to be executing is termed as a packet; this invention will facilitate instructions packed which has only the operands of instructions. (Instructions are operands only too.) Moreover, instruction packets size can vary in different machines which execute that same ISA. As in claim [11] above, if the speed of the active operations are many folds faster that passive operations, then this can be used in a scheme to execute many packets at one. Explicitly scheduling active instructions (active operations) can be used as a scheme to reduce dummy instruction mentioned in claim [3] which is a waste of computing power and time. The dummy

slot (or no operation slot) will be replaced by a scheduled active instruction. If the instruction packet size is static across different implementations of the same ISA, packets also can be schedulable. This scheduling can be done as an operand present in each instruction and in case if the packets are schedulable in each packet or can be implemented using a FU which is instructed to schedule a certain set of instructions or a set of packets. The operation will be explicitly scheduled by the compiler. In case the ISA does not have explicit scheduling or in case the schedules provided by the compiler need re-scheduling an embodiment can be used. This embodiment may introduce an instruction to the instruction pipeline to schedule an instruction or it might insert an extra operand or alter the existing operands for re-scheduling.

[19] As in claim [3] above where instructions include move and pair of select and target instructions, may include a scheme where, selection of a set of registers and subsequently moving them to an intermediate set of registers and then moving each register in the intermediate set of registers to the actual destination. The registers selected for moving could be both input or output registers. The intermediate set of registers may be dedicated or dictated by a controller.

[20] The mode of practice in claim [2] above and claim [3] above where instructions include operands would encompass a situation where the operands of the instructions are dynamically altered or inserted. In some cases a new instruction may be inserted into the instruction stream by inserting a complete set of operands to constitute a new instruction, between the two instructions.

[21] The apparatus in claim [1] above to claim [3] above may include explicit cache management operation instrumented through FUs. This also will enable the compiler to have finer control of the cache. The computing apparatus could include a circular cache for instructions. The execution takes place in ascending order of cached instructions. In the course of execution certain blocks of the cache may be replaced. The cache would be partitioned and the data will be placed in one of the partitions. These partitions can also be changed in size and absolute location. Different threads may also have different instruction partitions. After the partition boundary is reached execution begins at the other end of the cache partition. The partitions are a controlled window into the absolute cache. The instructions executed are from one of the activated partitions. Certain data partitions may overlap so that inter-process is possible.

[22] The scheme of claim [21] above would include situations where there is a window into the cache which sees the partitions and the data and instructions within the partitions in a given process context. When process context is changed then the window is changed and a new set of partitions will come into play. The data partitions can overlap to facilitate inter process communication.

[23] The apparatus in claim [1] above to claim [3] above where a computational task may be carried out, may include a scheme to switch the threaded context using the stack. The context information will be stored in the stack. Process switching can also be done in such a way; sharing of values between processes can be done using shared registers in a circular arrangement of registers, which are addressed through a window depending upon the context. In this implementation where the stack is used for thread

and process context switching, part of the stack could reside in a cache partition which is addressed in a circular manner and which is synchronized with the main memory.

[24] The method of claim [8] above where passive operations are scheduled, may include a scheme where the where registers are immunized from change after scheduling (by raising an error – possibly using interrupts, or completely ignoring the writes to the registers) or the operation would take the input register values at the point of the schedule been activated. Depending upon the mode of practice the register which holds the schedule can be made immune to changers like the above or the schedule can be continuously changed.

[25] The mode of practice as in claim [11] above where there are active and passive operations, would include facilitate a scheme where an active operation is broken in to operations like:

- Register read
- Register write.

If intermediate transfer registers are used, as in claim [19] then an active operation can be further broken down to include:

- Move to intermediate registers.

The passive operations will include:

- Compute FU outputs.

These can operations can all the executed in parallel.

[26] The apparatuses in the form of FU the scheme in claim [1] to [3] wherein the operations carried out by them inherently parallel, can be further used in implement operations that are traditionally carried out in interrupts. When coupled with the ability to load and unload FUs, operation in par with software interrupt can also be implemented using them.

[27] The when generating code for the apparatus in claim [1] where there are a finite number of FUs which has input and output registers; a slotted dependency graphs can be used. A slotted dependency graph is a dependency graph which is divided into level based on the execution cycle; and in each level there are a finite number of slots for the available FUs. Moreover, when generating code for the apparatus of claim [11] where an instruction is broken to smaller operation, each smaller operation can be modelled as a slot depending on the availability of operations. In this scheme, there are constrains which define that zero or more dependent operation should proceed and follow an operation. The operation available in a slot can be defines as static number for each operations; or as a group of numbers representing the operation in association with a state, where the state of preceding levels define the number of available operations in each level. Certain operations may alter the state. E.g. a stall in a class of instructions or scheduling instructions. If the number of operation for a particular level is not given it could default to a predefined value. This can be used where availability operation only change is special cases. Further more a limit can be placed on the number of certain types of operation. E.g. the maximum number of reads and the maximum number of writes in a level. Inclusion rules exclusion rules can also be specified where nodes representing certain types of instruction cannot appear in the same level. E.g. read operations and write operation. Special operations like schedule operation will make the scheduled operation unavailable for a specific number of levels. The scheduled operation slots

should be differentiated for the period of un-availability. If available number of operations possible in a given level exceeds what actually is available, dummy (no operation) operators can be inserted or scheduling operation can be done or stalling a class of operation can be done as in claim [11]. These operations can be done by the compiler or in hardware or both. Hardware dependency adjustments, including scheduling adjustments in claim [14] and claim [11], could include insertion of dummy instructions, rescheduling of operations and stalling of a class of instructions. But these need not be represented in the initial Slotted Dependency Graphs used by the compiler. Scheduling an active operation would include scheduling reads and writes. Constraint can also be defined for the maximum number of scheduled operations of a given type of operation or of a set of operations. If the number of possible operation of a certain class exceeds the number available, then these need to be moved to another level. When constraints are added to a Slotted Dependency Graph it can be called a Constrained Slotted Dependency Graph. In the compilation process first a dependency graph can be generated, then it could be transformed to a Slotted Dependency Graph which is intern transformed into a Constrained Slotted Dependency Graph. Depending on the type of contains applied, the Constrained Slotted Dependency Graph may also be generated in stagers by applying one of more constraints at a time to transform the graph to the next stage. The appropriate instructions are generated from the final graph.

[28] The scheme in claim [2] above where decoding is eliminated, can be also used as a scheme to: simplify the computing apparatus to save the number of components used thus also acting as heat reduction scheme, and reduce computing time of an instruction.

[29] The arrangement claims [1] and [2] where the FUs input parameters can be set by moving data into them would enable the compiler to have finer control over the computing apparatus. This could be used in schemes including but not limited to: simplification of hardware thus reduce components and heat dissipated; optimal use of hardware by executing all the possible operation in an algorithm in the maximum parallel manner only limited by the availability of FUs.

[30] The arrangement claims [1] and [2] would enable the compiler to do the hazard management in a computational apparatus. Since the compiler has finer control over the processing apparatus, the instructions could be aligned in such a way that hazards do not take pace.

[31] The practice in claim [30] where the compiler handles hazard management, could be used as a scheme to reduce components used for hazard management thus also reducing heat.

Abstract of Invention:

11 Abstract

[1] Implementation of a processor using and implied set of instruction would eliminate the time and resources consumed in the processing. In an implied architecture, there can be one or more implied instructions, which occur in pre-defined order. In the main architecture mentioned in this document (for illustration purposes), there are one (move instruction), two (Select and Target) or move instructions in recurring patterns, are used to achieve a computational task.